Behavioural Model-based Control for Autonomic Software Components

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier

Behavioural Model-based Control for Autonomic Software Components

Frederico Alvares and Eric Rutten
INRIA Rhône-Alpes
Email: {frederico.alvares,eric.rutten}@inria.fr

Lionel Seinturier
University of Lille 1 & INRIA Lille
Email: lionel.seinturier@inria.fr

***Abstract*—Autonomic Managers (AMs) have been largely used to autonomously control reconfigurations within software components. This management is performed based on past monitoring events, configurations as well as behavioural programs defining the adaptation logics and invariant properties. The challenge here is to provide assurances on navigation through the configuration space, which requires taking decisions that involve predictions on possible futures of the system. This paper proposes the design of AMs based on logical discrete control approaches, where the use of behavioural models enriches the manager with a knowledge not only on events, states and past history, but also with possible future configurations. We define a Domain Specific Language, named Ctrl-F, which provides high-level constructs to describe behavioural programs in the context of software components. The formal definition of Ctrl-F is given by translation to Finite State Automata, which allow for the exploration of behavioural programs by verification or Discrete Controller Synthesis, automatically generating a controller enforcing correct behaviours. We implement an AM by integrating the result of Ctrl-F compilation and validate it with an adaptation scenario over *Znn.com*, a self-adaptive case study.**

I. Introduction

Software architectures and more specifically software components have played a major role in the development of autonomic software systems. Besides the usual benefits of modularity and reuse [1], adaptability and reconfigurability are key properties which are sought with this approach: one wants to be able to adapt the component assemblies (configurations) in order to cope with new requirements and new execution conditions occurring at runtime [2][3]. Autonomic Managers (AM) can be designed in the form of feedback control loops [4], to take adaptation decisions at runtime by choosing the next configuration (e.g., a set of architectural elements) in function of not only the observed past history (monitoring events, states, configurations), but also on behavioural programs describing the adaptation logics and properties to be kept invariant all over the managed system's execution [5][6]. It may be non-trivial, however, especially for large and complex architectures (e.g., web applications with hundreds/thousands of replicated components with specific tunning parameters and constraints), to conceive well-mastered AMs, with guarantees on the autonomic behaviours, i.e., with assurances on the way the navigation through the configuration space is performed. In fact, that form of autonomic decision must involve not only updating the current advancement in the behaviors, but also some predictions on the possible futures of the system.

Those behavioural guarantees in the autonomic management can be achieved with the support of control theoretical approaches, where the use of behavioural models allows for predictive decisions. Control-based approaches for autonomic computing have been investigated, mainly concerning quantitative aspects and using continous control [7]. Since we are focusing on logical control for chosing configurations, we will consider discrete control. In particular, we address the design of such a *decision-maker* as a Discrete Controller Synthesis (DCS) problem [8], which consists in automatically generating a controller capable of controlling a set of input variables such that a given temporal property is satisfied. To that end, we propose the design of software component AMs based on Finite State Automata (FSA) behavioural models, which provide knowledge on events, states, past history as well as on possible futures, i.e., the space of reachable configurations. This way, we will be able to avoid going in behavioural program branches leading to wrong configurations.

We propose a Domain Specific Language (DSL), called Ctrl-F, to describe behavioural programs in the context of software components. Ctrl-F provides high-level constructs for the description of architectural reconfigurations and policies in the form of constraints. In this paper, we give the behavioural definition of Ctrl-F programs by the translation to a FSA model. More precisely, we provide full translation to the reactive language Heptagon/BZR [9], which allows the compilation towards formal tools and thereby benefit from exploration by both DCS and verification. As the decision-making can be very costly and exponential in the number of possible configurations, DCS generates controllers in an off-line manner, which means that the formal exploration of behavioural programs is compiled away. Furthermore, this is done in a maximal permissive way, that is, the controller keeps the maximum of possible configurations not violating the stated policies, and hence making the autonomic system maximally flexible. As a result, for a given behavioral program, we produce an executable function which, at each decision step of the AM, takes the current state and current events, and returns a control value which corresponds to the next configuration such that the stated policies are enforced. We illustrate our approach with the case study *Znn.com* [10] and show its applicability through an adaptation scenario.

II. Background

This section introduces concepts of Architecture Description Languages, underlying Ctrl-F, and Reactive Systems, which we use to ensure correctness of adaptation behaviours.

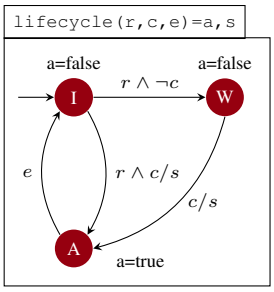
A. Architecture Description Languages

Software architectures define the high-level structure of software systems, by describing how they are organized by means of a composition of components [1]. Architecture description languages (ADLs) [11] are used to capture these architectures. Although numerous ADLs exist, the architectural elements proposed in almost all of them follow the same conceptual basis [12]. A *component* is defined as the most elementary unit of processing or data and it is usually decomposed into two parts: the implementation and the *interface*. The implementation describes the internal behaviour of the actual component, whereas the interfaces define how the component should interact with the environment. A component can be defined as simple or *composite* (i.e., composed of other components). A *connector* corresponds to interactions among components. Actually, it mediates an inter-component communication in diverse forms of interactions. A *configuration* corresponds to a directed graph of components and connectors describing the application's structure and/or a description on how the interactions among components evolve over time. Other elements like attributes, constraints or architectural styles may also often appear in ADLs [12]. In short, ADLs are usually utilized to define either static or initial architectural configurations, from which, by relying on introspection and reconfiguration [13], [2], one can add or remove elements at runtime.

B. Reactive Systemsand languages

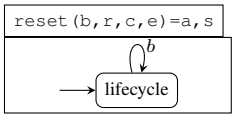
Reactive Languages have been proposed to describe systems that at each reaction perform a step taking input flows, computing transitions, updating states, triggering actions, emitting output flows [14]. Their definition is often based on Finite State Automata (FSA), which constitute the basic formalism for representing behaviours, as is the case of StateCharts [15] and of synchronous languages [16].

1) Heptagon: Heptagon/BZR [9] is an example of such languages. It allows the definition of reactive systems by means of generalized Moore machines, i.e., with mixed synchronous data-flow equations and automata [17]. An Heptagon program is modularly structured with a set of *nodes*. Each node corresponds to a reactive behaviour that takes as input and produces as output a set of stream values. The body of a node consists of a set of declarations that take the form of either automata or equations. The equations determine the values for each output, in terms of expressions on inputs' instantaneous values or other flows values. Figure 1 shows an Heptagon program in both graphical and textual representations. The program describes the control of a component's life-cycle that can be in either idle (*I*), waiting (*W*) or active (*A*) states. The program takes as input three boolean variables: `r`, which represents a request signal for the component; `c`, which represents an external condition (to be used later on as controllable variable); and `e`, to represent an end signal. It produces as output two boolean values, one that indicates whether the component is active (`a`) the another indicating a start action (`s`). When in the initial state, upon a request signal (i.e., when r is true), the automaton leads to either waiting or active states, depending whether the condition `c` holds. If it does not, it goes first to the waiting state and then to active when `c` becomes true. All the incoming transitions arriving at active state triggers the start action (`s`). From active state, it goes back to idle state upon an end signal.




```
node lifecycle(r,c,e:bool)
       returns(a,s:bool)
let automaton
  state I do
   a=false;s=r & c
   until r & c then A |
         r & not c then W
  state W do
   a=false;s=c
   until c then A
  state A do
   a=true;s=false
   until e then I
 end; tel
```

Fig. 1. Graphical and Textual Representation of Component Life-cycle.




```
node reset(b,r,c,e:bool)
     returns(a,s:bool)
let automaton
  state H do (a,s)=lifecycle(r,c,e)
          until b then H
  end; tel
```

Fig. 2. Example of Hierarchical Composition.

One important characteristic of Heptagon/BZR is the support for hierarchical and parallel automata composition. Figure 2 illustrates an example of hierarchical composition, in which a single-stated super-automaton embodies the *lifecycle* automaton. It has a self-transition that results in the resetting of the containing automata (i.e., *lifecycle*) at every occurrence of signal *b*. Listing 1 illustrates the parallel composition of two instances of the *delayable* node (and the operator ';'). They run in parallel, in a synchronous way, meaning that one global step corresponds to one local step for every node.

2) Contracts and Discrete Controller Synthesis: BZR is an extension of Heptagon with specific constructs for Discrete Controller Synthesis (DCS). That makes Heptagon/BZR distinguishable since its compilation may involve formal tools such as Sigali [8] and Reax [18] for DCS purposes. A DCS consists in automatically generating a controller capable of acting on the original program to control input variables such that a given temporal property is enforced. In Heptagon/BZR, DCS is achieved by associating a *contract* to a node. A contract is itself a program with two outputs: e_A, an assumption on the node environment; and e_G, a property to be enforced by the node. A set $\{c_1, c_2, \ldots, c_q\}$ of local controllable variables is used for ensuring this objective. Putting it differently, the contract means that the node will be controlled by giving values to $\{c_1, \ldots, c_q\}$ such that given any input flow satisfying assumption e_A, the output will always satisfy goal e_G. When a contract has no controllable variables specified, a verification that e_G is satisfied in the reachable state space is performed by model checking, even if no controller is generated.

Listing 1. Example of Contract in Heptagon/BZR.

```
1 node twocomponents(r1,r2,e1,e2:bool) returns
                      (a1,a2,s1,s2:bool)
2 contract
3 assume true
4 enforce not(a1 and a2)
5 with (c1,c2)
6 let
7  (a1,s1)=lifecycle(r1,c1,e1);(a2,s2)=lifecycle(r2,c2,e2)
8 tel
```

Listing 1 shows an example of contract on a node enclosing a parallel composition of two instances of *lifecycle* (cf. Figure

1). It is composed of three blocks. The *assume* block (line 3), which in this case, states that there is no assumption on the environment (i.e., $e_A = true$). The *enforce* block (line 4) describes the control objective : $e_G = \neg(a1 \wedge a2)$, meaning that both components are mutually exclusive, i.e., they cannot be active at the same time. Lastly, the *with* block (line 5) defines two controllable variables that are used within the node (line 7) : In practice they will be given values such that variables `a1` and `a2` are never both true at the same instant.

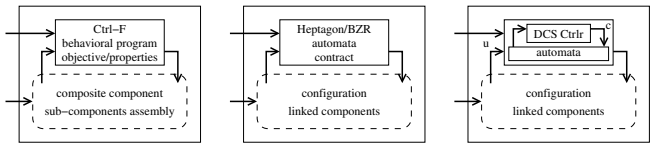
3) Compilation and code generation: The Heptagon/BZR compilation chain is as follows: from source code, the Heptagon/BZR compiler produces as output a sequential code in a general-purpose programming language (e.g., Java or C) implementing the control logic, in the form of a step function to be called at each decision in the autonomic loop. At the same time, if the code provided as input contains any *contract*, the compiler will also generate a intermediary code that will be given as input to the model checker (e.g., Sigali or Reax), which will, in turn, perform the DCS and produce as output an Heptagon/BZR code corresponding to the generated controller. The latter is then compiled again so as to have an executable code also for the generated controller.

III. Motivating Example: *Znn.com*

Znn.com [10] is an experimental platform for self-adaptive applications, which mimics a news website. As in any web application, Znn.com follows a typical client-server n-tiers architecture, meaning that it relies on a load balancer to redirect requests from clients to a pool of replicated servers. The number of active servers can be regulated in order to maintain a good trade-off between response time and resource utilization. Hence, the objective of Znn.com is to provide news content to its clients/visitors within a reasonable response time, while keeping costs as low as possible and/or under control (i.e., constrained by a certain budget).

There might be times where only the pool of servers is not enough to provide the desired Quality of Service (QoS). For instance, in order to face workload spikes, Znn.com could be forced to degrade the content fidelity so as to require fewer resources to provide the same level of QoS. For this purpose, Znn.com servers are able to deliver news contents in three different ways: (i) with high quality images, (ii) with low quality images, and (iii) with only text. Hence, content fidelity can be seen as another criteria. In summary, the objectives are as follows: (1) keep the performance (in terms of response time) as high as possible; (2) keep content fidelity as high as possible or above a certain threshold; (3) keep the number of active servers as low as possible or under a certain threshold. In order to achieve them, we may tune: (1) the number of active servers and (2) the content fidelity of each server.

We extend Znn.com by enabling its replication in presence of different content providers: one specialized in soccer and another one specialized in politics. These two instances of Znn.com will be sharing the same physical infrastructure (e.g., processing power, memory, etc.). Depending on the contract signed between the service provider and his/her clients that establishes the terms of use of the service, Znn.com Service Provider can give higher or lower priority to a certain client. For instance, during the World Cup the content provider specialized in soccer will always have priority over the other one. Conversely, during the elections, the politics-specialized content provider is the one that has the priority.



(a) Specification. (b) Model. (c) Execution.

Fig. 3. Approach Overview.

IV. General Framework

A. Approach Overview

Our approach consists in seamlessly conceiving autonomic component-based applications by relying on a high-level behavioural description. The principle is to have an AM embodying a feedback control loop within each component. The manager takes decisions in response to occurred events, while taking into consideration the current/past configurations, a behavioural program, and determining as result which configurations have to be terminated and which ones have to be started. We rely on Ctrl-F to specify: (i) behaviours in a process-like manner (in terms of sequences, alternative/conditional/parallel branches and loops of configurations); and (ii) policies, which take the form of properties that have to be kept invariant regardless of the configuration, as depicted in Figure 3(a).

The behavioural program defined by Ctrl-F provides the AM an extra level of knowledge on the possible futures of the component's configuration, that is, it enables the AM to explore the space of reachable configurations so as to avoid branches that may lead, in the future, to configurations violating the stated policies. To that end, we provide a set of translation schemes allowing for the automatic translation from a Ctrl-F description to the reactive language Heptagon/BZR and thereby benefit from DCS. There will be an Heptagon/BZR automaton and contract corresponding to the behavioural program and policies associated to each component under control, as can be seen in Figure 3(b). The Heptagon/BZR program, once equipped with contracts, allows us to either perform formal verification on the behavioural program with respect to the policies; and/or to obtain, via DCS, a correct-by-construction controller (cf. Figure 3 (c)). That is to say that the generated controller will be capable of controlling the automaton that models the component behaviour so as to prevent it going in branches leading to bad states (i.e., configurations that violate the policies). This whole process, from the Ctrl-F description to the Heptagon/BZR translation, is detailed next.

B. Ctrl-F Language

Ctrl-F language can be divided into two parts: a static architectural elements description; and a dynamic description of behaviours and policies, detailed more in Sections V and VI.

1) Architectural Concepts: The static part of Ctrl-F shares the same concepts of many existing ADLs (cf. Section II). In Ctrl-F, a *configuration* is defined as a set of *instances* of

components, a set of *bindings* connecting *server* and *client interfaces* of those *instances* (i.e., an assembly), and/or a set of *attributes* assignment to *values*. That is to say that a *configuration* is a snapshot of the *attributes* valuation, the current (sub) components *instances* within the concerned (super) component and the *bindings* connecting *interfaces* of these *instances*. For example, we can model Znn.com as a single-configuration component (*Main*), enclosing two instances of a component named *Znn*. As depicted in Figure 4, the *Znn* component comprises instances of components *LoadBalancer* and *AppServer*. A single instance of the former (*lb*) dispatches requests to one, two or three instances of the latter (*as1*, *as2* and *as3*). That makes three possible configurations: *conf1*, with only *as1*; *conf2*, with *as1* and *as2*; and *conf3*, with the three of them. These configurations are chosen at runtime, according to events warning overload (*oload*) or underload (*uload*) situations, which are emitted by the *LoadBalancer* component every time the number of requests per second exceeds a threshold.

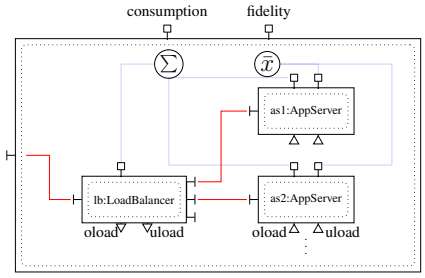


Fig. 4. Architecture Description of Znn.com.

While component *LoadBalancer* has only one attribute to indicate its consumption in terms of CPU percentage (e.g., 0.2 to express 20% of consumption), component *AppServer* has two attributes, one for the consumption level, just like *LoadBalancer*, and another to express its fidelity level: *text*, *img-ld* and *img-hd*. That is, *AppServer* may operate in three different configurations and the way attributes are valuated determines the current configuration. For example, we could have (*consumption=0.2, fidelity=0.25*), (*consumption=0.6, fidelity=0.5*) and (*consumption=1, fidelity=0.75*) to denote the configurations *text*, *img-ld* and *img-hd*, respectively. The value of those attributes can be propagated at the *Znn* component level: the consumption corresponding to the summation of the consumption of all (instances of) sub-components, while the fidelity level could be given by the average of all instances of *AppServer* defined within component *Znn*.

2) Behaviours: Behaviours in Ctrl-F are defined with the aid of a high-level imperative language, which consists of a set of behavioural statements (*sub-behaviours*) that can be composed together so as to provide more complex behaviours in terms of sequences of reconfiguration. In this context, a *configuration* is considered as an atomic behaviour, i.e., a behaviour that cannot be decomposed into other *sub-behaviours*. We assume that configurations do not have the capability to terminate or start themselves, meaning that they are explicitly requested or ended by behaviour *statements*. Hence, a reconfiguration occurs when the current configuration is terminated and the next one is started. Table I summarizes the behaviour statements of the Ctrl-F behavioural language.

TABLE I. SUMMARY OF BEHAVIOUR STATEMENTS.

Statement	Description
B **when** e_1 **do** B_1, ... , e_n **do** B_n **end**	While executing B when e_i execute B_i
case c_1 **then** B_1, ... , c_n **then** B_n **end**	Execute B_i if c_i holds
$B_1 \mid\mid B_2$	Execute B_1 and B_2 in parallel
$B_1 \mid B_2$	Execute either B_1 or B_2
do B **every** e	Execute B and re-execute it at every occurrence of e

During the execution of a given behaviour B, the *when-do* statement states that when a given event of event type e_i arrives the configuration(s) that composes B should be terminated and that (those) of the corresponding behaviour B_i are started.

The *case-then* statement is quite similar to *when-do*. The difference resides mainly in the fact that a given behaviour B_i is executed if the corresponding condition c_i holds (e.g., conditions on attribute values), which means that it does not wait for a given event. The *parallel* statement states that two behaviours can be executed at the same time. That is to say that at a certain point there are two independent branches of behaviour executing in parallel. The *alternative* statement allows to describe choice points between configurations, or between more elaborated sequential behavior statements. They are left free in local specifications and will be resolved in upper level assemblies, in such a way as to satisfy the stated policies, by controlling these choice points appropriately. Finally, the *do-every* statement allows for the execution of a behaviour B and re-execution of it at every occurrence of an event of type e. It is noteworthy that behaviour B is preempted at every occurrence of e. In other words, the configuration(s) currently activated in B is (are) terminated, and the very first one(s) in B is (are) started.

3) Policies: They are expressed with high-level declarative constructs for constraints on configurations, either temporal or on attribute values. In general, they define a subset of all possible global configurations, where the system should remain invariant: this will be achieved by using the choice points in order to control the reconfigurations. An intuitive example is that two components in parallel branches might have each several possible modes, and some of them to be kept exclusive. This exclusion can be enforced by choosing the appropriate modes when starting the components.

The specification of constraints on attributes is straightforward, since it consists in predicates and/or primitives of optimization objectives (i.e., maximize or minimize) on component attributes. Temporal constraints, on the other hand, take the form of predicates on the order of configurations, which might be very helpful when there are many possible reconfiguration paths (by either *parallel* or *alternative* composition, for instance), in which case the manual specification of such constrained behaviour may become a very difficult task. In order to ease the specification of such kind of constraints, Ctrl-F provides four constructs, as follows: `conf1` **precedes** `conf2`: `conf1` must take place right before `conf2`. It does not mean that it is the only one, but it should be among the configurations taking place right before `conf2`; `conf1` **succeeds** `conf2`:`conf1` must take place right after `conf2`. Like in the precedes constraint, it does not mean that it is the only one to take place right after `conf2`; `conf1` **during** `conf2`:

conf1 must take place along with conf2; conf1 **between** (conf2,conf3): once conf2 is started, conf1 cannot be started until conf2 has been terminated and conf3, in turn, cannot be started before conf1 has been terminated.

C. General FSA Model Structure

The component is the core of Ctrl-F description and can be modeled as an Heptagon/BZR node, as shown in Figure 5. The node takes as input external request (r) and end notification (e) signals, and a set of events $\{v_1, \ldots, v_k\}$, which corresponds to the event types the component in question (comp) listens to. As output, it produces a set of request (resp. end) signals $\{r_1, \ldots, r_m\}$ (resp. $\{e_1, \ldots, e_m\}$) for each configuration $conf_i$, for $i \in [1, m]$, defined within the concerned component. In addition, it also returns a set of weights $\{w_1, \ldots, w_l\}$, for the attribute valuation for each attribute in the component. The main node (comp in Figure 5) may contain a contract in which a set of controllable variables $\{c_1, \ldots, c_q\}$ (in the case there is any choice point such as a behaviour with an alternative statement) and the reference to the set of stated policies ($\{p_1, \ldots, p_t\}$) in order for them to be enforced by the controller resulting from the DCS. The details on how policies are translated are given in Section VI.

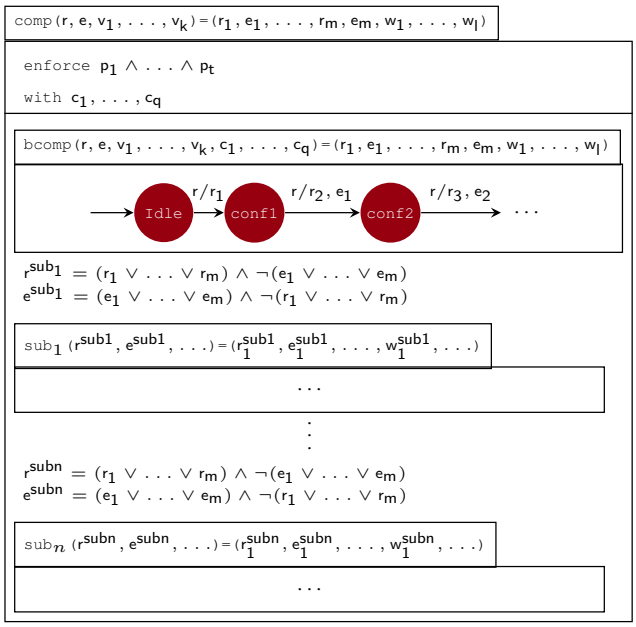


Fig. 5. Translation Scheme Overview.

Component behaviours are modeled as a sub-node (bcomp in Figure 5), which consists of an automaton describing the order and conditions under which configurations take place. For this purpose, it gets as input the same request (r), end (e) and event ($\{v_1, \ldots, v_k\}$) signals of the main node. As a result of the reaction to those signals, it produces the same signals for requesting ($\{r_1, \ldots, r_m\}$) and ending ($\{e_1, \ldots, e_m\}$) configurations as the weights ($\{w_1, \ldots, w_l\}$) corresponding to the attributes valuation in the current state (configuration) of the behaviour. We provide further details on the translation of behavioural statements in Section V. Lastly, there might also be some other sub-nodes ($\{sub_1, ..., sub_n\}$) referring to components instantiated within the concerned component, i.e., comp. They have interfaces and contents which are structurally identical to those of the main node. That is to say, that sub-nodes may have, in turn, a contract, a behaviour sub-node and a sub-node per component instance defined inside it. It is noteworthy that the request (r^{subi}) and end (e^{subi}) signals for a sub-component $sub_i \in \{sub_1, \ldots, sub_n\}$ are defined as equations of request and end signals. $\{r_1, \ldots, r_m\}$ and $\{e_1, \ldots, e_m\}$ are respectively the sets of request and end signals for the configurations $conf_1, \ldots, conf_m$ to which component sub_i belongs. That means that a sub-component sub_i will be requested if any configuration it belongs to is also requested ($r_1 \vee \ldots \vee r_m$) and none of them is terminated $\neg(e_1 \vee \ldots \vee e_m)$, which avoids emitting a request signal for an already active component. The same applies for its termination.

Listing 2 shows an excerpt of Heptagon/BZR model for components *Znn* (lines 6-23) and *AppServer* (lines 1-4). For node *appserver*, besides the request and end signals, it gets as inputs the events of type oload and uload (line 1). As output (lines 2 and 3), it produces request and end signals for configurations text (r_text and e_text), img-ld (r_ld and e_ld) and img-hd (r_hd and e_hd), apart from weights, i.e., attribute valuations (fidelity and consumption). Node znn has a very similar interface as *appserver*, except that it produces as output request and end signals for configurations *conf1* (r_conf1 and e_conf1), *conf2* (r_conf2 and e_conf2) and *conf3* (r_conf3 and e_conf3). Regarding its body (lines 10-22), znn comprises one instance of the node that models the behaviour (bznn, line 17) and three instances of node appserver (lines 18-20). The request and end signals for these instances can be derived from the request and end signals for configurations (lines 10-15). At last, attributes are values based on the values of attributes of the instances of node appserver (line 21).

Listing 2. Heptagon/BZR code for *Znn* and *AppServer*.

```
1  node appserver(r,e,oload,uload:bool) returns
2               (r_text,e_text,r_ld,e_ld,r_hd,e_hd:bool;
3                fidelity,consumption:int)
4  let ... tel
5
6  node znn(r,e,oload,uload:bool) returns
7          (r_conf1,e_conf1,...,r_conf3,e_conf3:bool;
8           fidelity,consumption:int)
9  let
10 r_as1 = r_conf1 or r_conf2 or r_conf3 and not(e_conf1 or
       e_conf2 or e_conf3);
11 r_as2 = r_conf2 or r_conf3 and not(e_conf2 or e_conf3);
12 r_as3 = r_conf3 and not(e_conf3);
13 e_as1 = e_conf1 or e_conf2 or e_conf3 and not(r_conf1 or
       r_conf2 or r_conf3);
14 e_as2 = e_conf2 or e_conf3 and not(r_conf2 or r_conf3);
15 e_as3 = e_conf3 and not(r_conf3);
16
17 (r_conf1,e_conf1,...) = bznn(r,e,oload,uload);
18 (r_text_as1,...,fid_as1,conso_as1) = appserver(r_as1,e_as2,
       oload,uload);
19 ...
20 (r_text_as3,...,fid_as3,conso_as3) = appserver(r_as3,e_as3,
       oload,uload);
21 consumption = conso_as1 + conso_as2 + conso_as3;
22 ...
23 tel
```

V. Behaviours

For each program in Ctrl-F, we need to construct a FSA model, in Heptagon/BZR, of all its possible behaviours. We translate each behaviour statement defined inside another behaviour as sub-automaton, hierarchically decomposing the whole behaviour into smaller pieces, down to a configuration.

A. The top-most behaviour

The top-most automaton i.e., the automaton modeling the whole behaviour consists of a two-state model, as depicted in Figure 6 (a). The automaton is in state *Idle* when the component does not take part in the current configuration. Upon a request signal (r), it goes to *Active* state, from where it can go back again to *Idle* state again upon an end signal (e). *Active* state accommodates a behaviour statement itself, which is itself modeled as a sub-automaton of state *A*.

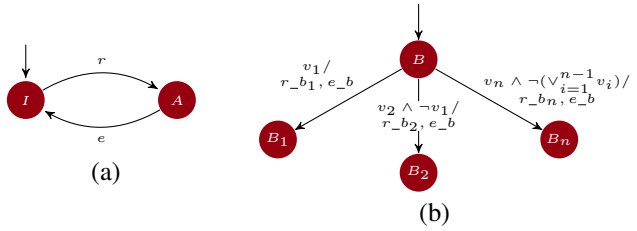


Fig. 6. FSA Modeling: (a) Lifecycle; (b) *When-Do*.

B. Statements

The automaton that models the statement *when-do* (cf. Figure 6(b)) consists of an initial state B corresponding to the first behaviour statement to be executed. The automaton goes to state B_i (corresponding to the execution of the next behaviour) upon a signal (event) v_i while producing signals for requesting the initiation of to the next behaviour (r_b_i) and the termination (e_b) the current one (for $1 \leq i \leq n$). It is important to notice that upon two events at the same time, a priority is given according to the order behaviours are declared. For instance, if v_1 and v_2 triggers, respectively, behaviours B_1 and B_2, then B_1 will be triggered if declared before B_2.

Both behaviour statements *case* and *alternative* can be modeled by the automaton shown in Figure 7. As the sub-behaviour statements should be executed at the very first instant upon the request of the *case* or *alternative* statement, the automaton must be composed in parallel with the automaton modeling the main behaviour (inside node `bcomp`, in Figure 5). Hence, a *case* or an *alternative* statement is modeled as a simple state inside the (super) automaton in the hierarchy that models the main behaviour. Upon a request to those statements (signal r), the main automaton emits a request signal r' that will trigger a transition from state W to the next state (B_1 or B_2) according to variable c. Then it can go either to another behaviour, if another r' is emitted and c states so; or back to W if an end signal (e') is emitted. There are two differences between the use of this automaton for a *case* or an *alternative* statement. First, for the *case* statement, several (i.e., more than two) branches are allowed, so there might be more states ($B_1, B_2, \ldots, B_n$) referring to each branch as well as their corresponding conditions $c_1, c_2, ..., c_n$, which was omitted here for readability reasons. Second, for the *alternative* statement, the conditions c_i will be considered as *controllable* variables in Heptagon/BZR. Thus, a DCS should be performed to guarantee that the stated policies are not violated.

The automaton model for the *do-every* statement is shown in Figure 8(a). It consists of a single-state automaton, which means that it starts by directly executing statement B. It has a self-transition at every occurrence of signal s, while emitting

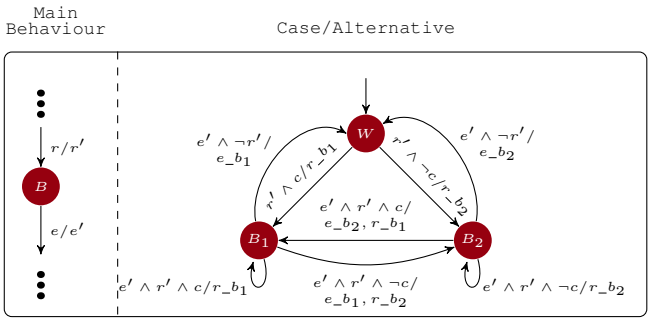


Fig. 7. Parallel Composition of Automata Modeling the Main Behaviour and the *Case/Alternative* Statement.

end (e_b) and request (r_b) signals, that is, statement B is re-executed at every occurrence of event s. Finally, Figure 8(b) presents the model for the *Parallel* statement: simply in the parallel composition of sub-automata.

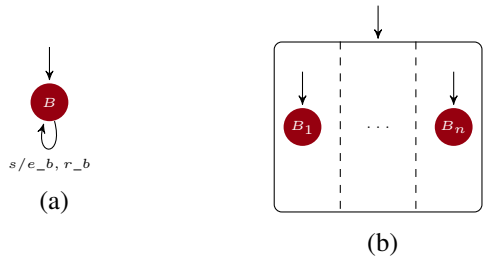


Fig. 8. Automata Modeling: (a) the *Every* and (b) *Parallel* statements.

C. Znn.com Example

We illustrate the use of the Ctrl-F statements for components *AppServer* and *Znn* (cf. Section IV-B1) as well as their respective translation into Heptagon/BZR. The expected behaviour for component *AppServer* is to pick one of its three configurations (*text*, *img-ld* or *img-hd*) at every occurrence of events of type *oload* (overload) or *uload* (underload). As can be seen in Listing 3, that behaviour can be decomposed in a *do-every* (line 3) statement composed of an *alternative* one.

Listing 3. Behaviour of AppServer Component.

```
1 component AppServer { ...
2   behaviour {
3     do text | img-ld | img-hd every (oload or uload)
4   }}
```

Regarding component *Znn*, the expected behaviour is to start with the minimum number of *AppServer* instances (*conf1*) and add one instance, i.e., leading to *conf2*, upon an event of type *oload*. From *conf2*, one more instance must be added, upon an *oload* event, leading to configuration *conf3*. Alternatively, upon an event of type *uload*, one instance of *AppServer* must be removed, which will lead the application back to configuration *conf1*. Similarly, from configuration *conf3*, upon an event of type *uload*, another instance must be removed, which leads the application to configuration *conf2*.

As shown in Listing 4, that behaviour can be achieved with a main *do-every* statement (lines 3-11), which executes a *when-do* statement (lines 4-10) at every occurrence of an

event of type *e1*. In practice, the firing of this event allows to go back to the beginning of the *when-do* statement, that is, the configuration *conf1* regardless of the current configuration being executed. According to the *when-do* statement, *conf1* is executed until the occurrence of an event of type *oload* (line 4), then another *do-every* statement is executed (lines 5-9), which in turn, just like the other one, executes another *when-do* statement (lines 6-8) and repeats it at every occurrence of an event of type *e2*. Again, that structure allows the application to go back to the beginning of the *when-do* statement, that is, the configuration *conf2*. Configuration *conf2* is executed until an event of type either *oload* or *uload* occurs. For the former case (line 6), another *when-do* statement takes place, whereas for the latter (line 7) a configuration named *emitter1* is executed. The *when-do* statement (line 6) consists in executing configuration *conf3* until an event of type *uload* occurs, then a configuration named *emitter2* takes place. Note that configurations *emitter1* and *emitter2* are special configurations that contain, each one, an instance of the pre-defined component *Emitter* (omitted here due to space constraints). The purpose of this component is to emit events such as the ones of type *e1* and *e2*. This allows the application to trigger the inner-statements within the *do-every* statements (lines 3-11 and 5-9) and thus be able to go backwards to configurations *conf1* and *conf2*, from configurations *conf2* and *conf3*, respectively.

Listing 4. Behaviour of Znn Component.

```
1  component Znn {...
2    behaviour {
3      do
4        conf1 when oload do
5          do
6       conf2 when oload do (conf3 when uload do emitter2 end),
7                   uload do emitter1
8          end
9        every e2
10     end
11   every e1
12 }}
```

Figure 9 illustrates the translation for the behaviours defined in Listing 3. It consists of a parallel composition of two automata: one to model the behaviour itself (on the left-hand side), and another to model the *alternative* sub-behaviour statement (on the right-hand side). The first automaton corresponds to the top-most automaton, as the one shown in Figure 6(a). The active state comprises a sub-automaton representing the *do-every* statement, which starts by state *B* and restarts it at every occurrence of events *oload* (overload) or *uload* (underload) while emitting at the same time request and end signals (r_b and e_b, respectively). The request signal (r_b) is used by the second automaton in order to enable transitions to states representing configurations (*txt*, *ld* and *hd*) according to the controllable variables c_1 and c_2, while emitting proper request signals (r_txt or r_ld) for the next configurations and end signals (e_txt or e_ld) for the current one. The end signal (e_b), on the other hand, is used to enable transitions to other or even the same configuration, in the presence of the request signal, or to the waiting state W, in the absence of the request signal. It should be mentioned that due to the lack of space, we omitted the outgoing and incoming transitions of state *hd* (configuration *img-hd*). In the generated executable code, the output of those automata will be connected to pieces of code dedicated to trigger the actual reconfigurations. For instance, the presence of signals r_ld and e_txt will trigger

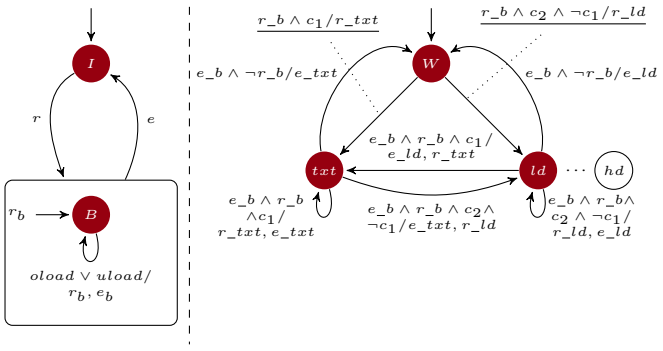


Fig. 9. Translation of the component *AppServer* behaviour.

the reconfiguration script that changes the content fidelity of given component from *text* to *img-ld* (cf. Section VII).

VI. Policies

A. Constraints/Optimization on Attributes

Constraints on attribute values can be translated into Heptagon/BZR in a very straightforward way. Indeed, they correspond to a set of boolean equations defined within the nodes that model components where the policies are stated. Then, the references to the equations are used inside the *enforce* block of a contract in order to state that they always hold by the control on the values of controllable variables. Listing 5 illustrates examples of constraints and optimization on component attributes. The first two policies state that the overall fidelity for component instance *soccer* should be greater or equal to 0.75, whereas that for instance *politics* should be maximized. Putting it differently, instance *soccer* must never have its content fidelity degraded, which means that it will have always priority over *politics*. The third policy states that the overall consumption should not exceed 5, which could be interpreted as a constraint on the physical resource capacity, that is, the number of available machines or processing units.

Listing 5. Example of Constraint and Optimization on Attributes.

```
1 component Main {...
2  policy { soccer.fidelity >= 0.75 }
3  policy { maximize politics.fidelity }
4  policy { (soccer.consumption+politics.consumption) <= 5}}
```

Listing 6. Example of Constraint on Attribute in Heptagon/BZR.

```
1 node main(r,e:bool;...) returns(...,p1:bool)
2 contract
3 enforce p1 and ...
4 with (...)
5 let ...
6    (...,soccer_consumption)=znn(...);
7    (...,politics_consumption)=znn(...);
8    p1=(soccer_consumpton + politics_consumption) <= 5
9 ... tel
```

For illustration, Listing 6 shows how the last policy of the *Main* component (Listing 5, line 4) is translated into Heptagon/BZR. This constraint is defined as an equation (line 8) that depends on the integer outputs `soccer_consumption` and `politics_consumption`, which are produced by the respective instances of node `znn` (lines 6 and 7). This equation is hence used in the enforce block of the contract (line 3).

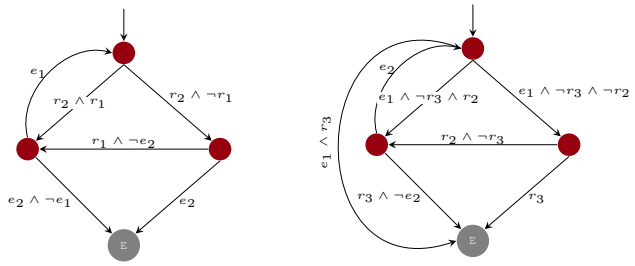
Although the declaration of optimization objectives are currently not supported by Heptagon/BZR, one may model a one-step optimization directly within the DCS tools Heptagon/BZR relies on [8] [18]. Please see [19] for more details.

B. Temporal Constraints

Temporal constraints refer to constraints on the logical order of configurations. They are modeled in Heptagon/BZR by a set of boolean equations of request (r) and end (e) signals that are emitted by automata modeling behaviours. For simple constraints like `conf1` **`succeeds`** `conf2` (resp. `conf1` **`precedes`** `conf2`), just a predicate like $e\_conf2 \Rightarrow r\_conf1$ (resp. $e\_conf1 \Rightarrow r\_conf2$) suffices. However, whenever there is a need for keeping track of the sequence of signals (to request and/or end configurations), the use of observer automata is needed. Observer automata are placed in parallel with the behavior automata, and generated in Heptagon/BZR as part of the contract. The principle is to have an automaton that observes the sequence of signals that leads to a policy violation and state that the state resulting from that sequence (an "error" state) should never be reached. Again, here we can rely on the enforce block of a Heptagon/BZR contract. The DCS objective is the invariance of the state set deprived of those where the variable error is true.

Figure 10(a) depicts an observer that models the policy during (`conf1` **`during`** `conf2`), where r_1 and r_2 (resp. e_1 and e_2) correspond to the request (resp. end) signal for configurations `conf1` and `conf2`, respectively. The error state (E) is reached if `conf2` terminates before `conf1` ($e_2 \wedge \neg e_1$) or if `conf2` terminates before `conf1` has started. The observer that models the constraint between (`conf1` **`between`** `(conf2,conf3)`) is depicted in Figure 10(b). Similarly, r_1, r_2 and r_3 (resp. e_1, e_2 and e_3) correspond to the request (resp. end) signal for configurations `conf1`, `conf2` and `conf3`, respectively. The automaton goes to the error state (E) whenever configuration `conf3` is started (r_3 is emitted) after configuration `conf2` (e_2), except when configuration `conf1` is started and terminated (r_1 and e_1) in the between.

Listing 7 shows an example of how to apply temporal constraints, in which it is stated that configuration *img-ld* comes right after the termination of either configuration *text* or configuration *img-ld*. In this example, this policy avoids abrupt changes on the content fidelity, such as going directly from text to image high definition or the other way around. Again, it does not mean that no other configuration could take place along with *img-ld*, but the *alternative* statement in the behaviour described in Listing 3 means that only *img-ld* must take place right after either *text* or *img-hd* has terminated.



(a) During (b) Between

Fig. 10. Observer Automata for Temporal Constraints.

Listing 7. Example of Temporal Constraint.

```
component AppServer {...
  policy { img-ld succeeds text }
  policy { img-ld succeeds img-hd }}
```

VII. Implementation

A. Ctrl-F Compilation Tool-chain

The Ctrl-F compilation tool-chain is depicted in Figure 11. As can be seen, the compilation process can be split into two parts: (i) the reconfiguration logics and (ii) the behaviour/policy control and verification. The reconfiguration logics is implemented by the *ctrlf2fscript* compiler, which takes also as input a Ctrl-F definition and generates as output a script containing a set of procedures leading from/to all configurations. The behaviour/policy control and verification is performed by the *ctrlf2ept* compiler, which automatically translates a Ctrl-F definition into a Heptagon/BZR program, according to translation schemes presented in Sections V and VI. The result of the compilation of an Heptagon/BZR code is a sequential code in a general-purpose programming language (in our case Java) comprising two methods: `reset` and `step`. The former initializes the internal state of the program, whereas the latter is executed at each logical step to compute the output values based on a given vector of input values and the current state.

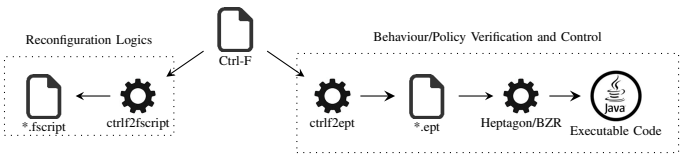


Fig. 11. Ctrl-F Compilation Tool-chain.

B. Autonomic Manager

The code generated by the Ctrl-F compilation is typically used by first executing the `reset` method and then by enclosing `step` and the reconfiguration scripts in an infinite loop, in which each iteration corresponds to a reaction to an event. As illustrated in Figure 12, upon the arrival of an event coming from the *Managed System* (e.g., oload and uload in the Znn.com example), the *Monitor* component triggers the *Analysis* component by invoking the step method. The result of the step method is a set of configuration request (r) and end (e) signals and based on these signals the *Plan* component finds the proper reconfiguration script to be executed by the *Execute* component, which delegates this task to a *Component Middleware* to interpret the script and then perform introspection and reconfiguration on the managed system's components at runtime. It is noteworthy that the Ctrl-F behavioural program allows the MAPE-K control loop to have *Knowledge* in terms of possible future configurations. In practice, after the Ctrl-F compilation, the knowledge is spread in the *Analysis* and *Plan* components. More concretely, the knowledge inside the *Analysis* consists of an automaton representation of the behavioural program, which is controlled by the DCS generated controller so as to enforce the stated policies. As the state-space exploration is very costly, it performed at design-time, whereas at runtime the controller is executed in a quasi-instantaneous way

by relying on the results of the off-line analysis. Regarding the *Plan* component, the knowledge corresponds to the set of all reconfiguration scripts. We implemented such an AM in FraSCAti [3], a middleware enabling runtime reconfiguration on Service Component Architecture applications [20], see [21].

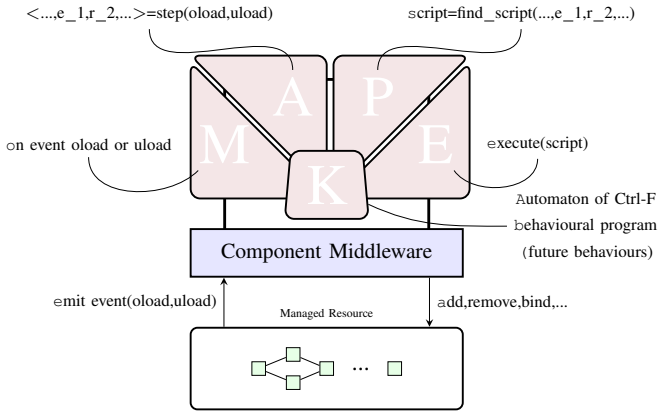


Fig. 12. MAPE-K Control Loop Wrapping the Ctrl-F Compiled Code.

C. Znn.com Adaptation Scenario

We simulated the execution of the two instances of *Znn.com* application, namely *soccer* and *politics*, under the administration of the AM presented in last section, to observe the control of reconfigurations taking into account a sequence of input events. The configurations are as defined in Section IV-B1, the behaviours for components *AppServer* and *Znn* are as stated in Listings 3 and 4, respectively, while the policies are as defined in Listing 5 and 7. as it can be observed in the first chart of figure 13, we scheduled a set of overload (*oload*) and underload (*uload*) events (vertical dashed lines), which simulate an increase followed by a decrease of the income workload for both soccer and politics instances. the other charts correspond to the overall resource consumption, the overall fidelity, and the fidelity level (i.e., configurations *text*, *img-ld* or *img-hd*) of the three instances of component *appserver* contained in both instances of component *znn*.

As the workload of *politics* increases, an event of type *oload* occurs at step 2. That triggers the reconfiguration of that instance from *conf1* to *conf2*, that is, one more instance of *AppServer* is added within the component *Znn*. We can observe also the progression in terms of resource consumption, as a consequence of this configuration. The same happens with *soccer* at step 3, and is repeated with *politics* and *soccer* again at steps 4 and 5. The difference, in this case, is that at step 4, the *politics* instance must reconfigure (to *conf3*) so as to cope with the current workload while keeping the overall consumption under control. In other words, it forces the *AppServer* instances *as2* and *as3* to degrade their fidelity level from *img-hd* to *img-ld*. It should be highlighted that although at least one of the *AppServer* instances (*as2* or *as3*) could be at that time at maximum fidelity level, the knowledge on the possible future configurations guarantees the maximum overall fidelity for instance *soccer* to the detriment of a degraded fidelity for instance *politics*, while respecting the temporal constraints expressed in Listing 7. Hence, at step 5, when the last *oload* event arrives, the fidelity level of *soccer* instance is preserved by gradually decreasing that of *politics*, that is, both instances *as2* and *as3* belonging to the *politics* instance are put in configuration *text*, but without jumping directly from from *img-hd*. At step 9, the first *uload* occurs as a consequence of the workload decrease. It triggers a reconfiguration in the *politics* instance as it goes from *conf3* to *conf2*, that is, it releases one instance of *AppServer* (*as3*). The same happens with *soccer* at step 10, which makes room on the resources and therefore allows *politics* to bring back the fidelity level of its *as2* to *img-ld*, and to the maximum level again at step 11. This is repeated again at steps 13 and 14 for instances *politics* and *soccer* respectively, bringing *consumption* to the same levels as in the beginning.

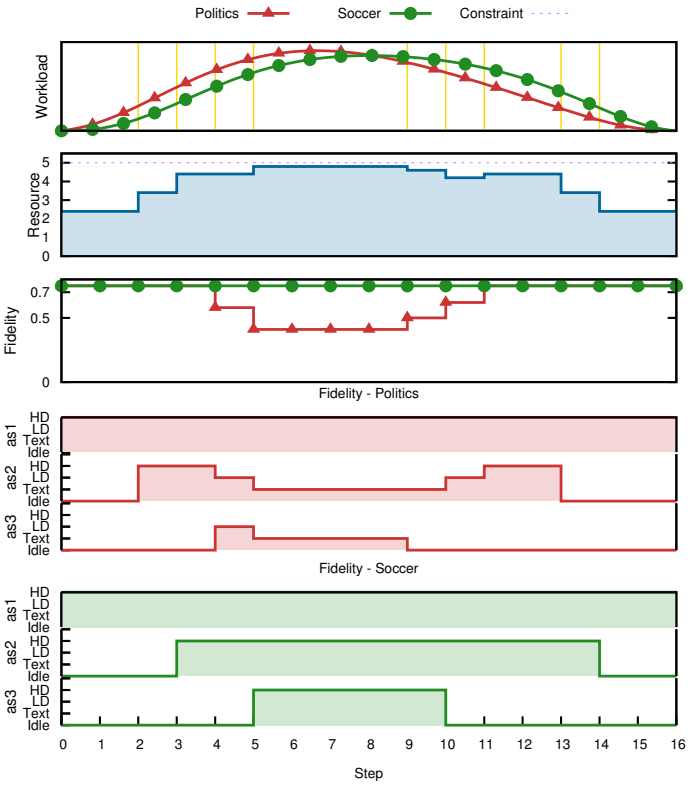


Fig. 13. Execution of the Adaptation Scenario.

The adaptation scenario is very useful to understand the dynamics behind an AM that is derived from a synchronous reactive program, which is in turn, obtained from Ctrl-F. Moreover, the scenario illustrates, in a pedagogical way, how controllers obtained by DCS are capable of controlling reconfigurations based not only on the current events and current/past configurations (states), but also on the possible future behaviours, that is, how controllers avoid branches that may lead to configurations violating the stated policies.

VIII. Related Work

Our work can be compared to a body of work in the domains of Component-based Software Development, Model-Driven Development and Control. In the domain of Component-based Software Development, Rainbow [5] provides an autonomic framework for Acme components [12]. A DSL called Stitch is used to express autonomic behaviours (called strategies) in a tree-like manner. Branches in strategies

are selected online by a utility-based algorithm according to runtime conditions. At the end (when it gets to a leaf in the tree), a strategy is evaluated as successful or failed and this information is used to improve the selection algorithm. Kouchnarenko and Weber [6] propose the use of temporal logics to integrate temporal requirements to adaptation policies in the context of Fractal components [13]. The policies specify reflection or enforcement mechanisms, which refer respectively to corrective reconfigurations triggered by unwanted behaviours, and avoidance of reconfigurations leading to unwanted states. While in those approaches, enforcement (resp. decisions over strategies' branches) and reflection are performed at runtime, in our approach, the decisional part of the AM is obtained in an off-line manner, through the reactive language Heptagon/BZR and by performing DCS. This way, the exploration of behavioural programs is compiled away, producing as result a maximal permissive and correct-by-construction controller that enforces correct autonomic behaviours. That can be seen as a tremendous advantage, since the formal exploration can be very costly and exponential in the number of possible configurations to be performed online, which is even more complex when the control is required to be least restrictive. Conversely, due to model incompleteness and uncertainties inherent to unpredictable environments, assumptions taken at design time may no longer hold at runtime. One way to mitigate this limitation is to have a multi-tier control, as proposed by D'Ippolito et al. [22]. The idea is that one can define multiple models and controllers associated to different levels of assumptions (from the least to the most restrictive) and guaranteeable functionalities. The level of control is then determined according to the validity of assumptions at runtime.

In [23][24], feature models are used to express variability in software systems. At runtime, a resolution mechanism is used for determining which features should be present so as to constitute configuration. Those approaches rely on Model-Driven Engineering to ease the mapping between features and architectures as well as to automatically and dynamically generate the adaptation logics, i.e., the reconfiguration actions leading the target system from the current to the target configuration. In the same direction, Pascual et al. [25] propose an approach for optimal resolution of architectural variability specified in the Common Variability Language (CVL) [26]. A major drawback of those approaches is that in the adaptation logics specified with feature models or CVL, there is no way to define stateful adaptation behaviours, i.e., sequences of reconfigurations. In fact, the resolution is generally performed based on the current state and/or constraints on the feature model. On the contrary, in our approach, in the underlying reactive model based on FSA, decisions are taken also based on the history and possible futures of configurations which allows us to define more interesting and complex behaviours, while providing guarantees on them.

As in our approach, in [27], the authors also rely on Heptagon/BZR and DCS techniques to model autonomic behaviours in the context of Fractal components. An et al. [28] used Heptagon/BZR to conceive AMs in the context of partially reconfigurable FPGAs (Field Programmable Gate Arrays). Although those approaches provide us with interesting insights on how adaptive behaviours can be formalized, there is no general method allowing for the direct translation from a high-level description (e.g., ADL) to a synchronous reactive model. It means that for each new application, the formal model has to be recreated. Moreover, reconfigurations are controlled at the level of fine-grained reconfiguration actions (e.g., add/remove components and bindings), which can be considered time-consuming and difficult to scale, especially for large-scale architectures. In comparison, Ctrl-F proposes a set of high-level constructs to ease the description of adaptation behaviours and policies of component-based architectures. In addition, we propose an extensible AM that bridges Ctrl-F and a real component platform. Delaval et al. [29] propose the use of components to embody AMs conceived with Heptagon/BZR. The idea is to have modular controllers that can be coordinated so as to work together in a coherent manner. The approach is complementary to ours: on the one hand, it does not provide means to describe behavioural programs for those managers, although the authors provide interesting intuitions on a methodology to do so. On the other hand, our approach does not provide means for the specification of the coordination among components' controllers. We do believe however that coordination is a major challenge that has to be tackled by any modular autonomic system. Hence, the integration of coordination aspects to Ctrl-F and its behavioural formalization must be considered in future work. Moreover, modularity seems to be an interesting perspective to mitigate the scalability issues due to state-space explorations.

IX. Conclusion

This paper presents a control-based approach for the design of Autonomic Managers (AMs) in the context of software components. In particular, we rely on Finite State Automata (FSA) models so as to provide AMs within components with predictive decisions on the possible futures of the component in question. We proposed Ctrl-F, a Domain-Specific Language for the description of behavioural programs, i.e., the adaptation logics and policies (properties to be kept invariant regardless of the configuration). We provide full translation from Ctrl-F programs to Heptagon/BZR, a FSA-based reactive language, which is integrated with formal tools allowing for exploration of behavioural programs by verification and Discrete Controller Synthesis. That is to say that we can provide behavioural guarantees by control, that is, by having automatically generated controllers enforcing behaviours such that policies are not violated. We applied Ctrl-F to *Znn.com*, a case study for self-adaptive systems, and showed how the result of its compilation is integrated within an AM. We validate our approach by executing such an AM in an adaptation scenario.

We are currently evaluating the use of Ctrl-F in case studies more realistic than *Znn.com*. We believe that we can apply our approach to other domains such as robotics and Cloud computing. For future work, we intend to pursue investigation on how our approach could be coupled with other forms of control (e.g., continuous control). We also plan to investigate aspects related to modularity and distribution of controllers, which may engender independence and asynchrony among controllers, especially if we want them to react to their environments at their own paces. This may give rise to conflicting behaviours, which makes it crucial to conceive appropriate high-level language constructs and mechanisms for the coordination among component managers and controllers. Prior work such as [29] seem to be a good start point.

REFERENCES

[1] I. Jacobson, M. Griss, and P. Jonsson, *Software reuse: architecture process and organization for business success*, ser. ACM Press books. ACM Press, 1997.

[2] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye, "FPath & FScript: Language support for navigation and reliable reconfiguration of Fractal architectures," *Annals of Telecommunications: Special Issue on Software Components – The Fractal Initiative*, 2008.

[3] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, 2012.

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[6] O. Kouchnarenko and J.-F. Weber, "Adapting component-based systems at runtime via policies with temporal patterns," in *FACS 2013, 10th Int. Symposium on Formal Aspects of Component Software, Revised Selected Papers*, ser. LNCS, J. L. Fiadeiro, Z. Liu, and J. Xue, Eds., vol. 8348. Nanchang, China: Springer, 2014, pp. 234–253, revised Selected Papers.

[7] T. Abdelzaher, Y. Diao, J. Hellerstein, C. Lu, and X. Zhu, "Introduction to control theory and its application to computing systems," in *Performance Modeling and Engineering*, Z. Liu and C. Xia, Eds. Springer US, 2008, pp. 185–215.

[8] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic, "Synthesis of discrete-event controllers based on the signal environment," *Discrete Event Dynamic System: Theory and Applications*, vol. 10, no. 4, pp. 325–346, October 2000.

[9] G. Delaval, H. Marchand, and E. Rutten, "Contracts for modular discrete controller synthesis," in *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010)*, Stockholm, Sweden, Apr. 2010.

[10] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, May 2009, pp. 132–141.

[11] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[12] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[13] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "An open component model and its support in java," in *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003)*, Edinburgh, Scotland, May 2004.

[14] D. Harel and A. Pnueli, "Logics and models of concurrent systems," K. R. Apt, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 1985, ch. On the Development of Reactive Systems, pp. 477–498.

[15] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[16] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Berlin, Heidelberg: Springer-Verlag, 2010.

[17] J.-L. Colaço, B. Pagano, and M. Pouzet, "A conservative extension of synchronous data-flow with state machines," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 173–182.

[18] N. Berthier and H. Marchand, "Discrete controller synthesis for infinite state systems with reax," in *IEEE International Workshop on Discrete Event Systems*, Cachan, France, May 2014, pp. 46–53.

[19] E. Dumitrescu, A. Girault, H. Marchand, and É. Rutten, "Multicriteria optimal reconfiguration of fault-tolerant real-time tasks," in *Workshop on Discrete Event Systems, WODES'10*. Berlin, Germany: IFAC, Aug. 2010, pp. 366–373. [Online]. Available: https://hal.inria.fr/inria-00510019

[20] "Service component architecture (sca)," http://www.oasis-opencsa.org/sca, accessed: 2015-01-29.

[21] F. Alvares De Oliveira Jr., E. Rutten, and L. Seinturier, "High-level Language Support for the Control of Reconfiguration in Component-based Architectures," INRIA Research Report, Research Report 8669, Jan. 2015. [Online]. Available: https://hal.inria.fr/hal-01103548

[22] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 688–699.

[23] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel, "Taming dynamically adaptive systems using models and aspects," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 122–132.

[24] C. Parra, X. Blanc, and L. Duchien, "Context awareness for dynamic service-oriented product lines," in *Proceedings of the 13th International Software Product Line Conference*, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 131–140.

[25] G. G. Pascual, M. Pinto, and L. Fuentes, "Run-time support to manage architectural variability specified with cvl," in *Proceedings of the 7th European Conference on Software Architecture*, ser. ECSA'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 282–298.

[26] O. Haugen, A. Wasowski, and K. Czarnecki, "Cvl: Common variability language," in *Proceedings of the 17th International Software Product Line Conference*, ser. SPLC '13. New York, NY, USA: ACM, 2013, pp. 277–277.

[27] T. Bouhadiba, Q. Sabah, G. Delaval, and E. Rutten, "Synchronous control of reconfiguration in fractal component-based systems: A case study," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: ACM, 2011, pp. 309–318.

[28] X. An, E. Rutten, J.-P. Diguet, N. Le Griguer, and A. Gamatié, "Autonomic Management of Dynamically Partially Reconfigurable FPGA Architectures Using Discrete Control," in *In Proc. of the 10th International Conference on Autonomic Computing (ICAC'13)*, SAN JOSE, CA, United States, Jun. 2013.

[29] G. Delaval, S. M.-K. Gueye, E. Rutten, and N. De Palma, "Modular coordination of multiple autonomic managers," in *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '14. New York, NY, USA: ACM, 2014, pp. 3–12.